Invariants and Recursion

Lecture 14

Today:

- Why Correctness is Important
- Invariants of Recursive Algorithms
- Running Time of Recursive Algorithms

Why Correctness is Important

Incorrect programs can be costly.

Famous Bugs:

In the early 1960s, one of the American spaceships in the Mariner series sent to Venus was lost forever at a cost of millions of dollars, due to a mistake in a flight control computer program.

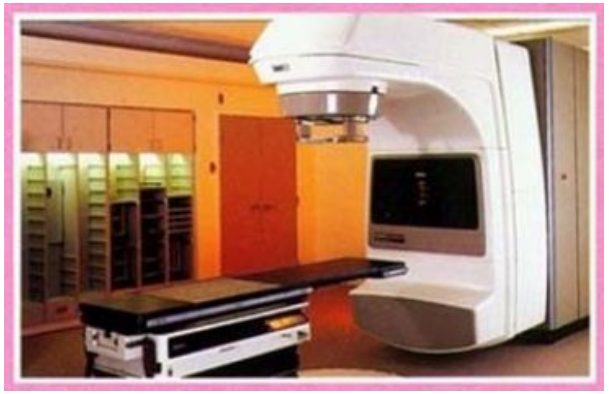
Mariner 1 Probe (1962)

Headline: “The most expensive hyphen in history.”

Famous Bugs

In a series of incidents between 1985 and 1987, several patients received massive radiation overdoses from Therac-25 radiation-therapy systems:

- three of them died from resulting complications.



Therac-25 (1986)

Therac-25 had been “improved” from previous models:

- the hardware safety interlocks from previous models had been “upgraded” by replacing them by software safety checks.

Famous Bugs

Some years ago, a Danish lady received, around her 107th birthday, a computerized letter from the local school authorities with instructions as to the registration procedure for first grade in elementary school.

- Program used two decimal digits to represent “age”.

This is similar in nature, but miniscule in scale in comparison, to the “Y2K bug”.

- Millions of programs used two digits for the year, assuming a “standard” 1900-prefix.

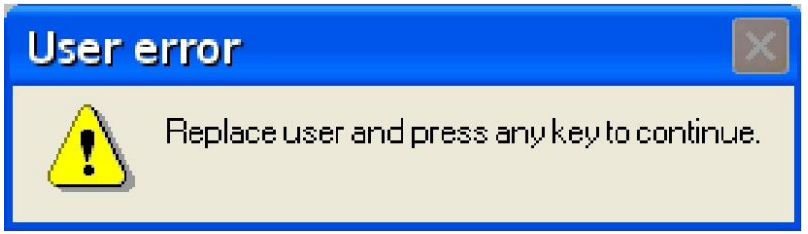
Q. What similar bug is coming soon?

Computers Do Not Err

Algorithms for computer execution are written in a formal unambiguous programming language

- Cannot be misinterpreted by the computer

Modern hardware is essentially bug-free. So, if our bank statement is in error and the banker mumbles that the computer made a mistake, we can be sure that it was not the computer that erred.



Either:

- incorrect data was input to a program; or
- the program itself contained an error

Similar programmer acronyms: PEBKAC RTFM

Preventing Bugs

Compilers:

- find syntax errors
- warn of common bugs + suggest fixes

Q. What warnings have you seen so far?

But beyond syntax, the compiler can't help you.

- Q. Why not?

Test your code to iron out the bugs!

Testing and Debugging

The more you test your program, the more likely you are to find bugs. Test sets can find:

- run-time errors
- logic errors
- infinite loops

But results are only as good as your test sets.

- Some bugs might never be discovered.
- Q. Is a test set as strong as a proof of a loop invariant?

Proving Correctness

Use mathematical proof techniques to reason about algorithms/programs.

- E.g., assertions and loop invariants.

Can we automate this proof process?

- Does there exist some sort of “super-algorithm” that would accept as inputs: a description of a problem P and an algorithm A, and respond `"yes"` or `"no"`?

In general, this is just wishful thinking: no such verifier can be constructed.

Loop Invariants (Review)

A loop invariant is a statement that is true every loop.

- usually asserted at the beginning of the loop
- usually parametrized by the loop index

A good loop invariant should indicate the progress of the algorithm

- the invariant should carry all state information, loop to loop.
- the invariant should imply the post-condition (the goal of the algorithm) at the end of the last loop.

Loop Invariants (Review)

Use mathematical reasoning to capture the behaviour of an algorithm:

- State *invariants* at various *checkpoints*.
- Show that the invariant holds:
  - Initialization: at the first checkpoint
  - Maintenance: during execution between checkpoints
- Conclude that the post-condition holds
  - Termination: the invariant holds at / after the last checkpoint

Q. This works pretty well for simple iteration, but what if your algorithm has no loops?

- Invariants are very powerful for recursive programs.

Invariants and Recursion

Rule of Thumb: You may assume the invariant holds for any *smaller* case.

```
//  Post:  Returns  n!
unsigned int fac(unsigned int n) {
    if (n <= 1) {
        return 1;
    }
    return n * fac(n-1);
}
```

Definition of Factorial

$0! = 1$

$1! = 1$

$n! = n \times (n - 1)!$, when $n \geq 2$

Recursive sub call

Assumes that `fac(n-1)` [correctly] returns `(n-1)!`

Another Similar Example

Recursive Definition of b^e

$b^0 = 1$

$b^e = b \times b^{(e-1)}$ when $e > 0$

```
//  Post:  Returns base**exp
int power(int base, unsigned int exp) {
   if (exp == 0) return 1;
   return base * power(base, exp-1);
}
```

Again, you are allowed to assume that the recursive sub call to `power(base, exp-1)`, a smaller case, returns the correct value.

Q. What does the running time depend on?

- It varies with the value of `exp`

Let N be the value of the parameter `exp`

- $T(N) = O(1) + T(N - 1)$ when $N > 0$
- $T(0) = O(1)$

A *recurrence relation*!

Solution: $T(N) = O(N)$

All Your Base Are Belong To Us

Can you do better?

- Use Divide and Conquer

Key Observation:

- Can you be quick if `exp` is even?
- Can call `power(base, exp/2)` and square the result.

Remember that *any* smaller case is correct.

- Not only an incrementally smaller case.

Divide and Conquer Solution

```
int power(int base, unsigned int exp) {
   if (exp == 0) return 1;
   int x = power(base, exp/2);
   if (exp % 2 == 1) {
      return x * x * base;
   } else {
      return x * x;
   }
}
```

Q. What's the running time?

- Again, let N be the value of `exp`
- $T(N) = O(1) + T(N/2)$ when $N > 0$
- $T(0) = O(1)$

Solution: $T(N) = O(\log N)$

Sorting by Recursion

Use Divide and Conquer to sort recursively.

1. Split the array into two roughly equal pieces.
2. Recursively sort each half.
   - This works because each piece is *smaller*.
3. Join the two pieces together to make one sorted array.

Two famous sorts behave this way: *mergesort* and *quicksort*.